Escaffre Thomas / El Hidaoui Azhar

Progress Report on HALProject Development

Introduction

In this report, we aim to summarize the progress we have made on our Hardware Abstraction Layer (HAL)project for the Atmega328p and Cortex-M3 microcontrollers. We will detail the steps we have taken, the challenges we faced, and the concepts we have learned throughout this process.

Project Overview

Our project involves developing HAL functionalities in Rust to provide modular and reusable interfaces for peripherals such as GPIO, USART, SPI, and I2C. Our objective is to ensure compatibility with embedded environments by adhering to the #![no_std] paradigm and using platform-specific implementations where required.

Key Steps in the Development

1. **Setting Up the Project Structure**
   We began by organizing the project into a structured directory, with separate modules for GPIO, USART, SPI, and later I2C. This modular architecture allows us to cleanly separate platform-specific code for Atmega328p and Cortex-M3 while maintaining a shared interface for each peripheral.

The structure looks as follows:

```
src/
├── gpio/            # GPIO module
│   ├── mod.rs       # Interface
│   ├── atmega328p.rs # Atmega328p-specific implementation
│   └── cortex_m3.rs  # Cortex-M3-specific implementation
├── usart/           # USART module
│   ├── mod.rs
│   ├── atmega328p.rs
│   └── cortex_m3.rs
├── spi/             # SPI module
│   ├── mod.rs
│   ├── atmega328p.rs
│   └── cortex_m3.rs
├── i2c/             # I2C module (recently added)
│   ├── mod.rs
│   ├── atmega328p.rs
│   └── cortex_m3.rs
├── main.rs          # Example usage
└── lib.rs           # Module exports
```

2. **Implementing Peripherals**
    - **GPIO**: The GPIO module was implemented first, providing functionality for configuring pins, writing values, and reading states. We directly manipulated memory-mapped registers, ensuring compatibility with #![no_std].
    - **USART**: We implemented USART support for both microcontrollers. This required understanding and configuring the respective UART interfaces, handling baud rates, and managing data transmission and reception.
    - **SPI**: SPI support was added next, involving initialization, data transfer, and ensuring the correct communication protocols for each board.
    - **I2C**: Finally, we added an I2C module. This was particularly challenging due to the more complex nature of I2C communication, requiring careful handling of start/stop conditions, address transmission, and data acknowledgment.
3. **Adapting to #![no_std]**
   A critical requirement for this project was ensuring all code was compatible with embedded environments lacking an operating system. We added #![no_std] to both main.rs and lib.rs and verified that none of our modules relied on std. This involved removing println! calls and replacing them with mechanisms suitable for embedded platforms, such as USART-based logging.
4. **Toolchain Configuration and Cross-Compilation**
   To target the Atmega328p and Cortex-M3 microcontrollers, we configured the Rust toolchain for cross-compilation:
    - For Cortex-M3, we used the thumbv7m-none-eabi target, ensuring compatibility with ARM-based microcontrollers.
    - For Atmega328p, we utilized a custom JSON target specification alongside avr-gcc for linking.
      Setting up .cargo/config.toml and ensuring the correct linker was a significant learning curve.

Challenges Encountered

1. **Understanding Embedded Concepts**
   As we progressed, we realized that writing efficient and hardware-specific Rust code required a deep understanding of the microcontrollers' datasheets, especially for register-level programming.
2. **no_std Compatibility**
   Ensuring our code was fully no_std required careful examination of dependencies and replacing default behaviors (e.g., panic handlers) with alternatives like panic-halt.
3. **Toolchain Setup**
   Configuring cross-compilation and linker behavior for both Cortex-M3 and Atmega328p involved troubleshooting several errors, such as missing toolchain targets and linker misconfigurations.

4. **Handling I2C Complexity**
   I2C was notably more complex than other peripherals due to the need for managing acknowledgments and timing.

Concepts Learned

1. **Embedded Rust Development**
   We became familiar with manipulating memory-mapped registers, understanding unsafe blocks, and writing modular HAL code.
2. **Cross-Compilation and Toolchains**
   Setting up Rust targets, configuring .cargo/config.toml, and understanding linkers were valuable lessons.
3. **Adapting to no_std**
   We learned how to write no_std code, manage panic handlers, and remove dependencies on std.
4. **Peripheral Protocols**
   We deepened our understanding of GPIO, USART, SPI, and I2C protocols, applying these concepts in hardware-specific implementations.

Conclusion

This project has been a rewarding journey, helping us gain expertise in embedded systems, Rust programming, and hardware abstraction. Despite the challenges, we have successfully implemented essential peripherals for two distinct microcontroller architectures, ensuring code modularity, reusability, and compatibility with no_std.